A Fiber Scheduler (C++) :: Assignment

If you have not already done so, complete the setup process for you choice of OS. Instructions for this are on Blackboard under Learning Materials.

All work for this assignment should be commited and pushed to a repo on Gitlab and a *README.md* included that details what is contained in the repo and your solution works and can be built and run. The URL for the repo should be submitted to BB.

The deadline for this assignment is 16th January 2025, by 14:00. The 48 hour grace period applies.

Introduction

The goal of this assignment is to implement a simple runtime that supports coperative tasks running within a single thread.

> POSIX or any other type of runtime threads cannot be used to complete this assignment.

Threads were designed to parallelize compute-intensive tasks, enabling independent components within a single application to progress on their own, while supporting 'simple' direct communication. However, there are many drawbacks to threads, in particular, they are both computational and memory intensive, which for applications that are I/O intensive or tasks that are short lived, for example, this can be performance bottle neck. Languages such as Javascript, used heavely in the I/O world of the web, have long avoided this through the use of callbacks and more recently *promises*.

The problem with callbacks, often refered to as 'callback hell', is the issue with nested callbacks, e.g. something like following:

```
asynFunc('whatever', function(err, data) {
  asynFunc('whatever', function(err, data) {
    asynFunc('whatever', function(err, data) {
      asynFunc('whatever', function(err, data) {
        asynFunc('whatever', function(err, data) {
          asynFunc('whatever', function(err, data) {
            // finally get around to do something
          });
        });
      });
    });
  });
});
```

Of course, there are ways of mitigating this somewhat, see for example, Node's use of streams.

Why does Javascript use callbacks? In Javascript callbacks can be either:

- Synchronous; A synchronous callback is executed direclty in as part of the evaluation of the function. In general, this is no different to passing a function to another function that can be used to compute values. C++, for example, supports lambda functions for this purpose
- Asynchronous; An asynchronous callback is executed after the execution of the function that uses the callback. This might happen a longtime in the future after the original function was called, e.g. once a network request has completed.

However, as already noted these days many applications are I/O (Input / Output) intensive and Javascript is not the only language of web. For a long time Microsoft have developed the language C# and pushed an alternative to the 'callback hell' of Javascript's callbacks, often refered to as co-routines or the similar concept Async Await. Async Await was not invented by the C# architects, it first appeared in Microsoft's F# language, but there is a strong argument that they help push them into the main stream developer's mind.

Async Await and Co-routines in one form or another are making their way into most modern programming languages, including Python, javascript, C++, Rust, and the Google's language Go is designed from the ground up around this notion.

> In truth there is a slight difference bettwen Async Await and Co-routines (as defined by D. Knuth), the details do not concern us here, but are something we will return to later in the module.

What is Async Await?

To understand Async Await we first need to explore the idea of a *Promise*, sometimes called Futures in other programming languages. Again we will use Javascript, but any of you that did the Internet of Things module will have already come across them in Python in the library *asyncio*, which we used for to communicate with our web sockets.

A Promise in Javascript is a guarantee that we will do something in the future, e.g. return the content from a web request. As in real live, a promise is either kept or it is not. In Javascript terms a Promise is either *resolved* when it is ready or it is *rejected*. This sounds a little like an if-statement, however, a promise is the result of asynchronous computation, fitting well with Javascript's asynchronous mindset, it does not wait for asynchronous operations to complete before processing the next sequential one, and a promise 'wraps' or 'defers' a computation until it is ready (resolved) or fails (rejected). Later a sequential statement can try to resolve the promise and use any returned results.

A promise has the following states:

- Pending: State at creation of the promise

- Resolved: A completed promise
- Rejected: Failed promise, which is an error state

So when a request is made to server via a promose, it is in a pending state until the data is returned.

A promise can be created as follows:

```
const aPromise = new Promise((resolve, reject) => {
    // condition
});
```

An (tivial) example use case is:

```
const aPromise = new Promise((resolve, reject) => {
    const condition = true;
    if(condition) {
         setTimeout(function(){
             resolve("Promise is resolved!"); // fulfilled
        }, 500);
    } else {
        reject('Promise is rejected!');
    }
});
```

Here you can see that is the promise is correctly resolved, then a call is made to the function *resolve*, while if it 'fails' then a call is made to 'reject'.

But how do we use a promise?

To use the above promise (`aPromise`) we use `then()` for *resolve* and `catch()` for reject.

```
aPromise.then((successMsg) => {
    console.log(successMsg);
}).catch((errorMsg) => {
    console.log(errorMsg);
});
```

This is syntax is a little 'noisy' but it can easily be wrapped in a function to help abstract out its use:

```
const wrapPromise= function() {
  aPromise.then((successMsg) => {
```

```
    console.log(successMsg);
  }).catch((errorMsg) => {
    console.log(errorMsg);
  });
}

wrapPromise();
```

Notice now that the above code avoids, for the most part, the complex control flow of using callbacks. Rather we can see that with some careful thought asynchronous control flow can be merged with synchronous code in a way that releaves much of the complexity. Of course, the use of then and catch still introduce quite a bit of additional syntax use, we must declare lambda functions, we have to handle both cases, which again introduces more noise and so on.

To avoid some of this synatatic complexity Async Await was introduced, which provides a synatatic wrapper for promises. It goes futher in intergrating asynchronous code with its synchronous counter part. Which for the most part makes it easier for us programmers to understand.

```
async function printAsync() {
  await printString("1")
  await printString("2")
  await printString("3")
}
```

Using the *async* keyword, on a function, effectly marks 'printAsync' as a function that returns a promise, with the use of the *await* keyword handling the *resolve* (`.then()`) process of a promise. It is important to note that to use *await* we must be within a context, e.g. function, marked with *async*. This implies that *await* cannot be used at the global level and those of you who remember using *asyncio* in Python will note that this was a similar restriction there, where a call to `asyncio.run(...)` was required. This is down to Javascript/Python runtimes requiring the ability to create an asynchronous execution context.

The Javascript *await* keyword is used in an *async* function, ensuring that all promises returned in the *async* function are synchronized, forcing them to wait for each other. This appoarch means that the callbacks `.then()` and `.catch()` are avoid. The value passed to the `.then()` callback is returned in place and `try/catch` blocks can be used to catch errors that would be handled by `.catch()`. The following is an example reworking of the above `wrapPromise` function:

```
async function wrapPromise() {
  try {
    let message = await aPromise;
    console.log(message);

  }catch((error) => {
      console.log("Error:" + error.message);
  })
}

// finally, call our async function

(async () => {
  await wrapPromise();
})();
```

If you are wondering how Async await are implemented, then you are in luck, as the remainding parts of this assignment look at how to implement a basic Fiber API in C++. We will not get as far as implementing promises and Async await, but it will support asynchronous functions, which we call tasks for this assignment. We will develop a round robin scheduler tasks and support giving up control from one task to allow another to run and so on. Later in the module we will look at how Async await can be implemented in a small language, building on the knowledge developed here.

> For those interested to see where a task runtime is used in practice, a much simplived version we are going to implement here, is used in productions systems, beyond just web requests and I/O, then check out Naughty Dog parallelized their game engine for PS4 and more recently PS5.

Before continuing it is worth reminding ourselves about the difference between preemptive and cooperative scheduling, which was covered in your 2nd year Operating Systems module:

- Preemptive scheduling is when the scheduling of the tasks is out of the control of the developer, entirely managed by a runtime. Whether the programmer is launching a synchronous or an asynchronous task, there is little difference in the code.
- Cooperative scheduling, the developer is responsible for telling the runtime when a task is expected to spend some time waiting for I/O or for another task, e.g. using the *await* keywork.

Tasks

There are 3 tasks and they will be marked as follows:

- Task 1, worth a maixmum of 40%
- Task 2, worth a maximum of 30%
- Task 3, worth a maximum of 30%

Each prior task must be attempted before moving on to the next.

Each task will be marked for:

- Funcitonality against the requirments
- Code quality
- Advanced techniques, e.g. template meta programming
- Testing
  - Use a unit test framework
- Documention in README.md

As noted in the introduction this assignment aims at implementing a simple runtime that supports execution of fibers. Why fibers? There are many other names used for tasks or fibers, including:

- green threads
- user-space threads
- coroutines
- tasklets
- microthreads

We are going to use fibers, for one as we are not planning on building a complete productiron runtime, which could provide addtional features that make it more of a fully fleged task runtime. For an example of such as runtime take a look at Intel's Thread Building Blocks. These kind of task runtimes have a lot of additional features that go beyond the lightweight user threads that we are discussing here.

As an example consider the following program:

```
void func1() {
    printf("fiber 1\n");

    fiber_exit();
}

void func1() {
    printf("fiber 2\n");

    fiber_exit();
}
```

```
int main() {
    fiber f2{func2};
    fiber f1{func1};

    spawn(&f1);
    spawn(&f2);

    do_it();

    return 0;
}
```

This program first creates two fibers, with the `spwan(...)` function, and then runs the scheduler, with `do_it()`, which returns only once all fibers has completd. Compiling and running this program produces the following output:

```
fiber 1
fiber 2
```

Functions such as `yield()` that relinquish control from a running fiber are emitted from this example, but form a key part of the ability for fibers to run concurrently. The full API that you will implement to complete this assignment is as follows:

```
//-----------------------------------------------------------------------
// API for fibers
//-----------------------------------------------------------------------

/// @brief terminates a fiber
///
///        note call control flow within a fiber must terminate with a call
///       to fiber_exit. Returning from a fiber without a call fiber_exit is
///        undefined.
void fiber_exit()();

/// @brief get pointer to data passed as part of fiber creation
/// @return pointer to data passed in at fiber creation
void * get_data();

/// @brief yield control back to scheduler
```

```
///       once rescheduled fiber will restart directly following call to
//        yield.
void yield();

/// @brief create a new task for execution
///        if called from outsite a task it will add task to run queue, but
///        nothing will happen until run() is called.
///
///        if called from within a task itself, then a new task is added to
///        the run queue and will execute when it is reached by the
///        scheduler, no need to call run().
///
/// @param function fiber execution body
/// @param data pointer acess from running fiber
void spawn(void (*function)(), void * data = nullptr);

/// @brief run with the current set of fibers queued.
///
///        returns only when all fiber have completed.
///
///        Calling do_it within a fiber is undefined.
void do_it();
```

Task 1

You might remember from your 1st year architecure or 2nd year operatings Systems modules that a thread running on a CPU core has some state that it requires to execute correclty, e.g.:

- Instruction Point (IP) that contains the address of the next instruction.
- Stack Pointer (IP) that contains the address of the thread's top of stack.
- Set of registers, used to pass/return values directly bettween function calls, and for intermidate calulations, e.g. operands to operations such as the CPU's ADD instruction.

The actual set of registers provided by a given CPU is, of course, defined by the architecture, e.g. x86-64 or Arm, but how they are mapped for executable programs is defined by the Operating System's Application Binary Interface (ABI). This will be different for different operating systems and archiectures. As the remote server is a Linux x86-64 system we need ABI for System V, originally developed by AMD when they move x86 to 64-bit and later adopted by Intel for 64-bit Linux systems.

According to Wikipedia In computer software, an application binary interface (ABI) is an interface between two binary program modules. Often, one of these modules is a library or operating system

facility, and the other is a program that is being run by a user. The key take away is that it allows interopbility within and between programs and the OS. But why does this matter to us?

Each fiber has its own stack and multiple fibers will run concurrently on a single thread. A thead has a single stack pointer and thus to support switching between threads we will need to provie the abilty to save a restore a fibers state. Most importantly a fiber's state will include the stack and instruction pointers, plus some additional registers that must be preserved for a fiber to work correctly when restored.

A small library is provided to support these features with the following API:

```
1  struct Context {
2    void *rip, *rsp;
3    void *rbx, *rbp, *r12, *r13, *r14, *r15;
4  };
5
6  extern "C" int get_context(Context *c);
7  extern "C" void set_context(Context *c);
8  extern "C" void swap_context(Context *out, Context *in);
```

It is implemented in assembly, as it is not possible to directly access the low-level machine even from C/C++. The library can be downloaded from Gitlab repo. It is recommended that you clone and then copy the header and assember file to where ever you are doing this work. Make sure you include them in your submited repo.

But what do these functions do? For now we will look at the first two. `get_context` saves the current state of execution so that it is possible to return to this point. `set_context` restores a previously saved context, returning execution to the point where the original call to `get_context` would have returned, i.e. the line after it in the program.

The following pseudo code shows this in action:

```
1 set x to 0
2 set c to get_context
3 output "a message"
4 if x == 0:
5   set x to x PLUS 1
6   call set_context with c
```

Setup a variable to indicate that the if has been executed, then saves the state at line 3, outputs a message, and then the first time through x is equal to 0, so the execution continues at line 6, adding

1 to x, then at line 7 control is restored to the previous state and execution starts again at line 4. This time at line 5 x does not equal 5, and control continues with out executing the if's body at line 8.

The program outputs:

```
a message
a message
```

Task 1 is to implement this program in C++, using the provided `context` library. You should mark the variable (x) with the keyword 'volatile' to avoid the compiler performing an optimization that would cause the conditional at line 5 to alway be true.

> it is fine to use a single git repo for all the tasks in this assignment, but do make sure that each file is clearly named and documented.

The pseudo code above transfered control within the same block in main. For fibers to work we need to transfer control to a function representing the fibers body and additionally we need to setup and switch the stack, as a fiber has its own stack and does not share one with the main thread. Implement, in a new file, the following pseudo code:

```
1  func foo:
2   output "you called foo"
3   # cannot return to main as they have different stack
4   call function exit
5
6  func main:
7   # allocate space for stack
8   data is an array of 4096 characters
9
10  # stacks grow downwards
11  sp is a pointer to characters
12  set sp to be data PLUS 4096
13
14  create an empty context c
15  set rip of c to foo;
16  set rsp of c to sp;
17
18  call set_context with c
```

What happens when you run this? It might work, but it is just as likely to fail! That's becuase our stack setup code is not valid!

For this to work we must account for Sys V ABI when it comes to stack alignment and layout. In particular, it states that the stack must be aligned to 16-bytes. This can be achevied by modifying sp after it is setup with the pseudo code:

```
1 set sp to sp AND -16L
```

> Not bitwise operations cannot be applied to pointers and so you C++ code will need to cast it to a `uintptr_t` and then back to a `char *`.

Finally, SyS V has some very specific rules about the stack when calling functions. This states that a 128-byte space is stored just beneath the stack pointer of the function, called the Red Zone, must be accounted for. This is done with the following pseudo code:

```
1 set sp to sp MINUS 128
```

Add these changes to your code and it should now work 100% of the time.

Extend your example to include a 'fiber' `goo`, which is 'called' using set_context and has it's own stack. `foo` must still run too. Where will control to `goo` happen?

Can you think of some interesting variants of using set/get_context?

Task 2

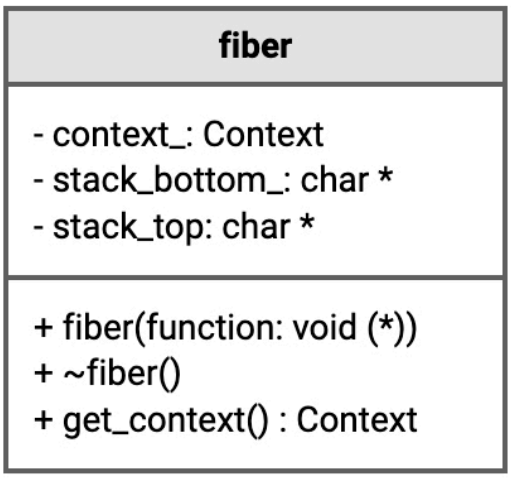
For the most part the 2nd example of Task 1 treated the functions `foo` and `goo` as if they were fibers. From this we can derive that a fiber has the following properties:

- a context which includes
  - RSP a pointer to the top of its stack (correctly aligned as per Sys V ABI)
  - RIP a pointer to a function that is its execution body

These two properties can be set when the fiber is created.

> RSP and RIP are constant for fibers that do not support yield. We will look at implementing yeild in the next task and for now we will focus on fibers that run to completion.

So it is clear that a class representing fibers should abstact over these properties, providing an interface to create fibers and access its properties. The following UML class diagram provides a simple represenation of this:



Implement a `class fiber` and rework the `foo` example of task 1 to make use of this. The following pseudo code should help guide you:

```
# implementation of fiber class
class fiber:
  ...

func foo:
  ...

func main:
  set f by creating fiber with foo

  set c calling method function get_context from f
  call set_context with c
```

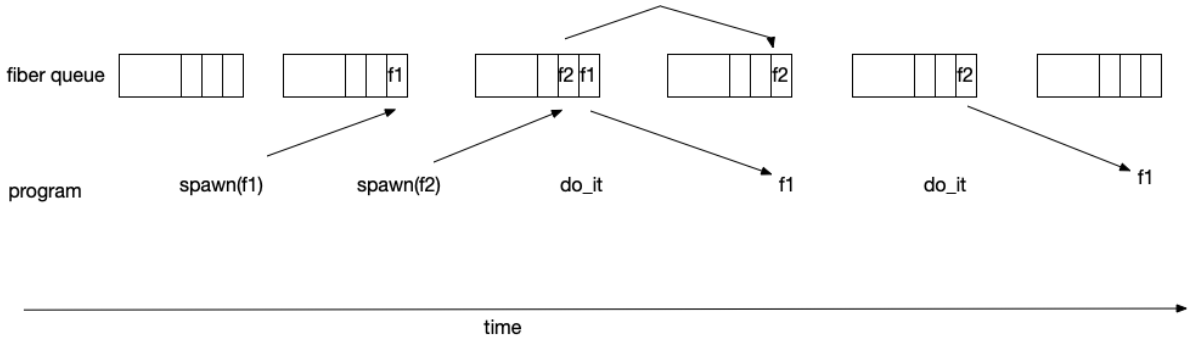
Now we want to extend our system to introduce a scheduler. The scheduler will enable fibers to be added to a queue, using `spawn`, and then these fibers can be executed in a round robin fashion with the method `do_it`.

As mutliple fibers can be spawned, but only one fiber runs at a time, the scheduler must keep track of these fibers waiting to run. To keep things simple they are processed in order of submission and one fiber is executed after the other, in a so called round-robin schedule. For now once a fiber starts executing it runs until it has completed, thus a fiber is queued and scheduled to run once all fibers queued before it have completed. As this is a queue, with first in first out (FIFO) semantics, the scheduler will need to use a data-structure that can store fibers in this way. For this we will use C++'s `std::deque`

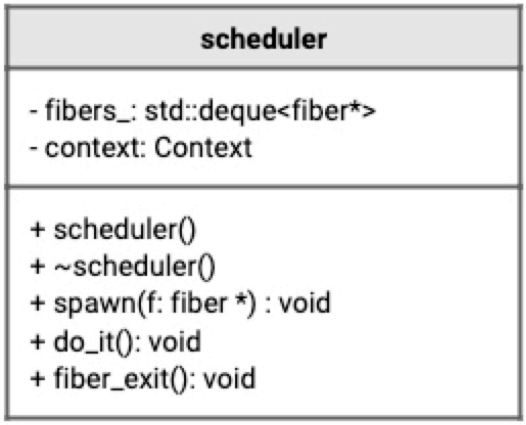
from the standard template library.

> Information on how to use `std::deque` can be found here.

A simple execution of the scheduler is viualised in the following diagram:



This scheduler class is represented by the following UML diagram:



The following pseudo code provides an outline of the class members:

```
class scheduler:
  property queue of fiber pointer fibers_
  property Context pointer context_

  method spawn with a fiber f:
    push f to back of fibers_

  method do_it:
    # return here to re-enter scheduler
    call get_context with context_

```

```
12      if fibers_ is not empty:
13        set f by poping from of fibers_
14
15        # jump to task
16        set c calling method function get_context from f
17        call set_context with c

18    method fiber_exit:
19        # jump back to the scheduler 'loop'
20        call set_context with context_
```

The interesting key elements are the methods `do_it` and `fiber_exit`, which together form the basis of the scheuler's while loop:

- Line 10 sets the entry point of the loop;
- Line 12 tests to see if the loop should continue, i.e if there are more fibers to execute;
- Line 17 transfers control to the next fiber; and finally
- Line 20 transfers control from the currently executing fiber back to the scheduler, which starts execution again at line 11, i.e. following where the context was saved.

Implement the scheduler class.

Before developing a test example for the new class you need to provide implementations for the functions `spawn`, `do_it`, and `fiber_exit`, as defined in the API in the introduction. These simply call into the global scheduler instance, `s`.

> note that we can't simply return at the end of the fiber, as a functions return address is stored on the stack, but remember a fiber has its own stack and set_context does not set a return address! In fact as this has to be the last value pushed to the stack it would be hard for us to do this implicity without compiler support.

Once completed use the following pseudo code to develop an example test case:

```
1  func func1:
2     output "fiber 1"
3     call fiber_exit
4
5  func func2:
6    output "fiber 2"
7    call fiber_exit
8
```

```
9  set s to be scheduler
10
11 func main:
12    set f2 to be fiber with func2
13    set f1 to be fiber with func1
14
15    call s method spawn with address of f1
16    call s method spawn with address of f2
17
18    call s method do_it
```

Why did we decide to make s global?

Try developing some other examples and test cases for your scheduler. Once done document your solution in the README.md.

To conclude this task consider the following pseudo code:

```
1  func func1:
2      output "fiber 1"
3      set dp to get_data
4      output "fiber 1: " *dp
5      set *dp to *dp PLUS 1
6      call fiber_exit
7
8  func func2:
9      set dp to get_data
10     output "fiber 2: " *dp
11     call fiber_exit
12
13 global s is set to scheduler
14
15 func main:
16     set d to 10
17     set dp to address of d
18     set f2 to be fiber with func2, dp
19     set f1 to be fiber with func1, dp
20
21     call s method spawn with address of f1
22     call s method spawn with address of f2
23
24     call s method do_it
```

```
25
```

Note that fibers are now created with a second argument, which is a pointer to an integer. Additionally, the fibers themselves us the the function `get_data` to access the pointer that was passed in at creation. Running this program should produce the following output:

```
fiber 1: 10
fiber 2: 11
```

So fiber 1 has updated the integer value that is visible to both fibers via the shared pointer. Extend your `scheduler` class to take an optional pointer argument for data. Implement the function `get_data`, which should return the pointer passed into the fiber when created or `nullptr` if no pointer was passed in. Implement the above pesudo code as an example test case.

Task 3

So far fibers are run to completion before returning control back to the scheduler via a call `fiber_exit`. As discussed in the introduction it is often useful for fibers to pause or yield control back to the scheduler before completing. This might simply to allow other fibers to get a shot at running, remember there is no support for preemptive scheduling, but other applications for this include the producer consumer pattern, where one fiber is the producer and one the consumer.

Implement support for `yield`. No pseudo code is provided for this, but the following can be used to develop a test example:

```
1  func f1:
2      output "fiber 1 before"
3      call yield
4      output "fiber 1 after"
5      call fiber_exit
6
7  func f2:
8      output "fiber 2"
9      fiber_exit
10
11 func main:
12    set f2 to be fiber with func2
13    set f1 to be fiber with func1
14
```

```
15      call s method spawn with address of f1
16      call s method spawn with address of f2
17
18      call s method do_it
```

Develop an example that demostrates the use of `yield` in combination with `get_data` and sharing data between fibers.

Before submitting your code it should be split into mutiple files, with your examples including header files for the fiber API, but all API implementation code should be in there own source files. For example, your compile line might look like:

```
clang++ -std=c++11 ex1.cpp fibers.cpp context.o
```

Finally, don't forget to document your soloution in README.md. Please take a look at projects on Github to see what a good readme might consist of. It is not just a blob of text and should include examples of code, how to build and run, screen shots, etc.